(4)

A Common Prototyping Language
Phase I

Final Report

Submitted to

Defense Advanced Research Projects Agency
Contracts Management (CMO)
1400 Wilson Blvd.
Arlington, VA 22209-2308

by

Michael Karr (Co-Principal Investigator)
Software Options, Inc. (Prime Contractor)
22 Hilliard Street
Cambridge, Mass. 02138

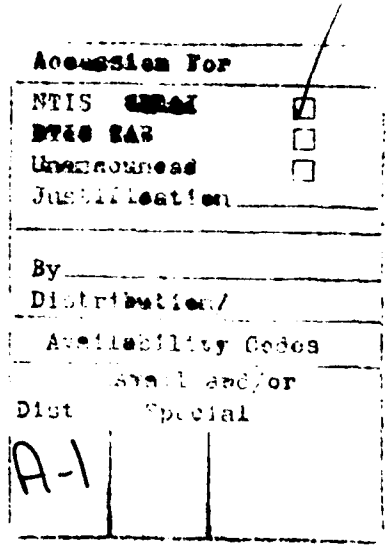
Paul Hudak (Co-Principal Investigator)
Yale University
New Haven, Conn. 06520

Robert M. Balzer
Information Sciences Institute
University of Southern California
Los Angeles, Calif. 90292

William E. Carlson
Intermetrics, Inc.
Cambridge, Mass. 02138

January 5, 1993

Contents



1 Introduction

The SOYII team (Software Options, Inc., Yale University, Intermetrics, Inc., and University of Southern California—Information Sciences Institute) has sought a broad, far-reaching, positive change in software engineering practices. Our technical goal has been, and is, to make disciplined experimentation and prototyping a *planned* part of the software lifecycle and to endow these processes with powerful tools and a powerful environment.

This immodest goal is plausible in light of the ProtoTech community that has emerged and the strength of the overall DARPA/ISTO software program. Our contribution has been compatible with this view. We (expecially Bill Carlson of Intermetrics) have devoted time and effort to solidifying the ProtoTech community and to helping others in the community have more impact—by challenging their thinking, suggesting opportunities for synergy, pointing out problems that need solutions, and suggesting technology transfer opportunities. Our technical contributions have largely been to provide infrastructure—for language definition and implementation, for environment development and tool integration, and for process description and experimentation. We have also developed specific instances of each—a language, an environment, and a software development process—that we summarize in the following sections, following which is a bibliography of available reports related to the project.

2 Languages and Language Families

The ProtoTech goal is to create languages that allow executable models (prototypes) to be expressed more precisely (formally), comprehensibly, and concisely than can be done with existing programming languages. An order of magnitude improvement in quality and productivity can be achieved by using innovative prototyping languages and tools being developed by ProtoTech contractors and the DSSA program.

2.1 Language Implementation Infrastructure

As languages become domain-specific, and so less general, the DoD will have more languages to support all of its applications. Such a situation will be practical only if there is a corresponding, and almost revolutionary, change in the technology that is used to implement languages and multilingual environments.

In recognition of this problem, a ProtoTech goal is to create technology for implementing multilingual environments, in which the languages share compiler components and tools, such as debuggers, optimizers, and performance analyzers. The kernel-based language implementation technology promoted by the SOYII team[KH91] can be a solution to the problem. Members of a *kernel* family share compiler components, including not just the syntax processors but, more importantly, the semantics-based translators and transformation machinery. Kernel-based implementations also share a theoretical foundation, that relies in part on model-based interpretation. The underlying theory simplifies the sharing of semantics-based tools and simplifies interoperability.

Software Options used the kernel-based approach in the definition and implementation of

a language. No theoretical or practical problems were encountered, but we had insufficient resources to apply it to other languages and make it easier to use by a broader community.

A major goal of the SOYII team was, and is, to make this kernel-based approach to language family implementations a practical reality. It will then be convenient and cost-effective to create and support a variety of languages that are specialized, and thereby well-suited to particular aspects of a total system. Users benefit from interoperability, allowing efficient composition of components written in different languages. The implementation effort required to add another surface language to an existing family becomes person-weeks instead of person-years, because each added language inherits an interpreter, an incremental compiler, sophisticated transformations and optimizations, a debugger, a performance analyzer, and other tools to form a complete state-of-the-art environment.

2.2 A Protytyping Language

As an instance of a suitable prototyping language, the SOYII team has based its work on the functional language Haskell[Pet92]. Prototyping in Haskell is most appropriate when developing prototypes from formal specifications, but Haskell is also an effective general purpose programming language. At Yale most of the work on prototyping technology has centered on Haskell in one way or another. The following paragraphs outline the pertinent results.

2.2.1 The Yale Haskell Implementation

Two compilers for Haskell were implemented, the first being a prototype that led to the econd (an almost complete rewrite of the first). The latest system was released in August 1992 and has been made freely available via internet ftp. It is estimated that about 50 groups are using the system world-wide. It is a very robust system that has been used in the classroom as well as for research purposes. It is considered to be the most complete Haskell implementation currently available.

2.2.2 MHDL and Parametric Type Classes

On a different project, researchers at Intermetrics have been designing MHDL, a Microwave Hardware Description Language[mhd92a, mhd92b, mhd92c]. In doing so, they have adopted many of the essential ideas in Haskell as a basis. One of the most important ideas adopted is Haskell's *type classes*, an effective approach to overloading that has an object-oriented feel. The Yale researchers have had fruitful collaborations with the Intermetrics group regarding the MHDL design. In fact, one of the extensions proposed to Haskell as part of the SOYII contract—so called *parametric* type classes—turned out to be exactly what the MHDL group needed.

Haskell's type classes have also influenced several other language designs, including Griffin and Rapide, two other languages being designed in the ProtoTech community. Both of these languages have features that closely resemble type classes.

2.2.3 Functional State

One of the most important problems addressed by the Yale group is that of incorporating state into a functional language. This is not only important in gaining more wide-spread acceptance of functional languages for use in the "real-world" but also in the general use of formal methods in software development.

At Yale several key results regarding state have been achieved in recent years. The first is the development of a *single-threaded type system* to ensure the proper threading of state objects in programs. If such objects are single-threaded, they can be implemented via conventional imperative techniques. The second is the development of the notion of a *mutable abstract datatype*. The key result here is that there is a non-trivial class of ADT's for which new operations (based on either *monads* or *continuations*) can be derived that guarantee single-threaded behavior *without* the use of a fancy (and usually complex) type system. This is a very significant—and surprising to many—result. It subsequently led to a further development, a conservative extension of the lambda-calculus called λ_{var}, that includes assignment yet retains referential transparency. This is also a surprising result.

Both mutable abstract datatypes and λ_{var} have been implemented in Yale Haskell. Several non-trivial programs involving both contiguous state objects, such as arrays, and linked objects, such as graphs, have been implemented with these extensions with very satisfying results: imperative-program efficiency is achieved in a purely functional language.

2.2.4 A Common Haskell/XX/YY System

The original SOYII proposal described the implementation of a common Haskell/ML/Scheme system to be built on top of the Standard ML of New Jersey (SML/NJ) implementation. Unfortunately, this effort ran into unforeseen problems not under the control of researchers at Yale. The primary difficulty was that the SML/NJ implementation was not stable. In particular, early discussions (prior to the proposal) with the SML/NJ implementors indicated that the intermediate language was to be stabilized, standardized, and exposed to interested users of the system. This was a critical building block for the Yale effort, but unfortunately it has not yet materialized.

Because of these problems, a different strategy was developed: the design and implementation of an integrated Haskell/Common-Lisp/XX system that will run on any Common Lisp platform. The "XX" is intended to be an imperative language such as Ada or C. This new strategy is especially appropriate for the Yale implementors, since the new compiler is written in Common Lisp. Even ignoring the interoperability issue, a key advantage of this approach is portability: the system should run wherever Common Lisp runs. Indeed, the current system is now running on 4 different Common Lisp implementations on a variety of machines.

2.2.5 Language Interoperability

Given a Common Lisp implementation platform, it would seem that interoperability of Haskell and Common Lisp programs would be fairly straightforward. From a bit-level perspective, this is indeed true. But from the language-level, it is not as easy. The main problem

is retaining referential transparency within Haskell in the presence of calls to Common Lisp (or other imperative language) procedures. Fortunately, the progress made on functional state mentioned earlier can be used as a basis for solving this problem as well (since the state is encapsulated in the Common Lisp program). The Yale researchers have developed a general method for interfacing to Common Lisp programs in a purely functional manner, and the method is currently being implemented. To demonstrate the feasibility of this approach, an interface to ISI's *relational abstraction* package written in Common Lisp is being developed and is planned for use as a demonstration of key ProtoTech technology.

2.3 A Prototyping Language Extension

Relational abstraction is a diction developed by ISI for software specification, useful in prototyping (and software migration). It presently is implemented as an extension of Common Lisp, a language frequently used for prototying, and, under separate funding, is being added to C++. Relational abstraction is complementary to other modern abstraction mechanisms and can be overlayed on top of them. It utilizes a typed entity-relationship data model that expresses relationships between typed values defined and created by the host language. Relations are used to associate these values with each other and can be used to access the set of such values that satisfy particular properties (i.e. queries) including "pointer following" navigation between values. To the extent that values in the host language have internal structure (such as the fields of a record), that structure is redundantly described via relations so that the individual parts of this structure can themselves be manipulated via relational abstraction.

Relational abstraction contains a number of mechanisms that help users understand what their prototype is doing, ensure that its behavior is consistent with their hypotheses, and reset its state so that new or more detailed experiments can be run.

- *Program Visualization* Relational abstraction provides associative query based access to data that augments the pointer-chasing accesses normally found in programming languages. This allows users to access descriptively data of interest. Futhermore, it initiates processing based on state changes (i.e. event triggers), thereby supporting data monitors that constantly maintain an updated view of the data they are displaying. Instrumentation of a prototype is handled similarly.

- *Hypothesis Checking* All state transitions occur within transactions, and before one is committed a set of user supplied consistency conditions is checked. These consistency conditions may be part of the application itself—checking the integrity of its data and signaling an exception or making repairs when a violation is detected—or they can be dynamically supplied as user hypotheses about the prototype's behavior that are constantly checked.

- *Roll-Back/Restart* The transaction history enables the prototype's behavior, in the form of its state transitions, to be unwound. This allows users to reset the prototype to a prior state, such as that existing prior to running an experiment, so that another experiment can be run.

ISI developed optimization technology for relational abstractions enabling their use in software destined for real-world applications. This technology is based on human-supplied *annotations* that guide the optimizations selected during a fully mechanical reduction process that reexpresses the prototype in the underlying host language so that it can then be compiled by that language's compiler.

Relational abstraction is a natural complement to a kernel-based approach to language implementation, although they exist at very different levels. Relational abstraction provides a set of high-level user-oriented abstractions and associated support technologies directly useful for prototyping and migration. A kernel-based approach to language implementation, on the other hand, provides the internal architecture and support for making them, or other useful capabilities, available throughout a family of languages—that is, for creating shared infrastructure. Thus, relational abstraction is an example of a particular set of capabilities that could be useful in many different languages; a kernel-based approach to language implementation is the means by which such capabilities can, in fact, be shared by multiple languages.

3 Environment Support and Tool Integration

Another ProtoTech community goal is to create powerful tools and environments for prototyping. Consistent with this goal was our work on the Artifacts System, an open-architecture environment that includes support for tool integration. The work makes the Artifacts System usable by a broader community: it is now more robust and practical for simultaneous use by 6-10 people and for projects involving hundreds of thousands of lines of source code.

The Artifacts System is a repository for all of the objects, called artifacts, that are developed in the course of a project and, as such, is responsible for automated change and configuration management of these objects. An artifact contains some piece of project data, in somewhat the same way that a file would hold such data. One of the important differences between artifacts and files is that the contents of an artifact are immutable. Another is that the contents of an artifact may have embedded references to other artifacts, and that the system understands these references. These properties have a profoundly simplifying effect on configuration management, tool invocation, and multi-user support.

In addition to its role as repository for project data, the Artifacts System is also responsible for tool invocation and for recording the results of tools, called "derivatives". The Artifacts System is extensible, having several axes of extensibility.

- *Artifact Type* The type, such as "C source text", "Lisp module", or "LaTeX document", of an artifact describes the structure of its contents.
- *Editor* A user prepares the contents of an artifact with an editor, currently with the Gnu EMACS/Epoch editor.
- *Deriver* A deriver is a tool capable of producing one or more derivatives from an artifact of a given type.

- *Derivative Class* The class, such as "executable file for the SPARC architecture" or "PostScript form of an activity description", of a derivative describes its structure.

Extending the set of derivers is the means by which tools are incorporated into the system, where by "tools" we mean compilers, typesetters, program analyzers, and the like. A deriver is often just a wrapper for a existing tool—it obtains inputs for the tool from the given artifact, invokes the tool, and arranges for output of the tool to be stored as derivatives of the artifact.

Extensions to the set of artifact types and/or derivative classes are often made in conjunction with an extension to the set of derivers. The Artifacts System imposes no restrictions on where or in what form the contents of artifacts or their derivatives are stored. When an extension to the set of artifact types is made, the extension connects an "artifact cell" (the small amount of information about the artifact known to the Artifacts System) to the "artifact contents" (the information which the artifacts hold). The connection between cell and contents must be consistent among the editors that can prepare, edit, or display artifacts of the type in question and among the derivers that take artifacts of that type as input.

Similarly, an extension to the set of derivative classes must specify the connection between the (short) string used to identify a given kind of derivative of a particular artifact and where the contents of the derivative is actually stored. The Artifacts System does not impose any restrictions on where or in what form the contents of the derivative is stored.

In short, extensions along the axes of artifact type, editor, derivative class, and deriver combine to provide a uniform interface between the Artifacts System and tools. The Artifacts System takes care of all aspects of change and configuration management, without understanding where the vast bulk of data is actually stored, and tools consume their input and record their output without any understanding that they play a role in the Artifacts System. Further, derivers can be written in any language, because the system simply spawns executables and communicates with the resulting process, completely oblivious to what language(s) might be used in obtaining that executable. Thus, the Artifacts System meets the goal of supporting a multiplicity of tools and languages.

During Phase I Intermetrics and Software Options experimented with the integration of a commercial Ada compiler[AR91]. The goal was to extend the benefits of the Artifacts System to Ada programmers. Though the experiment did not produce a usable Ada Artifacts System, it did result in much-improved facilities for extenders and an awareness of an opportunity to create a common framework for language artifacts.

Another result of Phase I, related to environments and their interoperability, is the paper on data transport[Kar92a], which grew out of the extensive discussion about a module interface formalism.

4 Process Support

A community goal is to provide the means to specify, implement, evolve, and evaluate prototyping-based development processes. ISI, in conjunction with the Process Working Group, has worked towards developing a formalism in which to define process meta-models; common process support for the definition, instantiation, and enactment of process models

expressed in those meta-models; generic support for distribution, persistence, and object management; and independence from the object manager being used to describe the product(s) produced.

The goal is to facilitate and accelerate the community's ability to support and experiment with diverse process formalisms. There are three stages to this effort, each of which has value in and of itself:

- understand the differences between process formalisms and the support they require,
- identify commonalities among those formalisms and their support, and
- construct sharable process infrastructure that provides some or all of those common capabilities.

The Process Virtual Machine is the name given to the sharable process infrastructure constructed during the third stage of this effort. It may take the form of a set of callable subroutines, a more structured architecture with explicit and implicit capabilities built into that architecture (i.e. a virtual machine upon which applications can be built), or a combination of the two. A design for the Process Virtual Machine is being prepared for review by the Process Working Group and the broader process community. If a consensus is reached on that design, an implementation is envisioned as part of Phase II.

Relevant to the ProtoTech goal, though under separate contracts, Software Options has been implementing a system, the Activity Coordination System, based on the process formalism developed in [Kar90]. The goal of the formalism and the system is to address issues that arise in activity coordination, including linguistic means for describing activities, the user interface, history, simulation, summarization, and cutover (changing of descriptions while activities based on them are under way).

Underlying the implementation of the Activity Coordination System are two fundamental issues: long-running execution and wide area communication. By "long-running execution", we mean the illusion that a program executes over a period of months or years. Since it is unreasonable to expect that any given machine will be up continuously for that length of time, and, since in activity coordination applications, executing programs spend most of their time waiting rather than computing, it is reasonable for a waiting execution to "sleep", i.e., to store its state safely in a database and cease being a process in the operating system sense. Sending a message to one of these long-running executions "awakens" it, i.e., starts an operating system process, retrieves the stored state from the database, and resumes computation.

To support long-running execution, we track the various ongoing executions, indicating whether they are asleep or awake and, if asleep, what program is to be run when its execution is awakened. The functionality supports initiating long-running executions, sending messages to such executions (awakening them first, if required), putting an awake execution to sleep if so requested, and terminating an execution. In all of this, note the irrelevance of the language in which the program underlying the execution is written—the generic machinery simply spawns an executable. It is equally ignorant of the content of the messages—it acts only as a router. Further, the database in which an awake execution chooses to save its state

before going to sleep is up to the program: when it goes to sleep, it hands the long-execution machinery a short string (e.g., a file name or a uid in an objectbase), indicating where it is stored. This string is given back to the program when it is awakened. There are no restrictions on where or in what form the state of a long-running execution is stored. The long-execution subsystem is completely open with respect to language and database.

Because the long-execution facility underlies the Activity Coordination System, that system inherits its openness with respect to language and database. Briefly, this happens as follows. An *activity description* is a high-level program, which may be written in a graphical notation, using nodes and arcs to describe a certain pattern of activity. There is a graphical editor[Kar92c, Kar92b] for constructing activity descriptions. Such a graph may be *instantiated*, i.e., an execution of it may be started. For expository purposes, it is convenient to think of an instantiation of an activity description as initiating concurrent long-running executions, one for each node. The executions for different nodes may be at different sites of a wide area network, i.e., they may be scattered all over the world. The executions communicate with each other as indicated by the wires (arcs) in the activity description. It is this concurrently long-running set of communicating executions that provides the desired activity coordination.

One axis of extensibility of this system is that of primitive activity descriptions. If the set of existing primitive activity descriptions is not sufficient—because, for example, the existing definitions do not allow access to a certain database that one wishes to use in an activity description—it is possible to add a new node definition. Adding a new primitive node definition requires writing some functions in some programming language, but not, as the reader can probably guess, in any particular language, because of the fact that they ultimately reside in a long-running program. Further, these functions can transact with whatever database they choose. The net result is the justification of the above claim that the Activity Coordination System inherits from the long-execution facility its openness with respect to language and database.

A long-execution mechanism by itself does not constitute an activity coordination facility. As we mentioned earlier, there is also the issue of communication between nodes of the graph, i.e., between long-running executions, and there is the issue of the visibility of the state of each node, which reduces to an issue of the communication of values, although not in an entirely naive way. A simple type system (itself extensible, although we don't pursue that further here)[Mor92] defines the interfaces for the functions that must be written when defining a new primitive activity descriptions. The interface between this type system and any given language is simple to implement, and, once this is done, there is no impediment to writing activity coordination functionality in the language of ones choosing. The existence of this language-independent interface is another piece of insurance that the system is multilingual.

The ISPW-6 software development process example[KC90] is being implemented with the Activity Coordination System. That, along with the smaller examples currently available[Mor92], provides one means for evaluation. The Process Virtual Machine, when it becomes well-defined, will provide another. The real goal, however, is for the Activity Coordination System to be used by a broader community during Phase II.

References

[AR91] M. Askanas and D. Rosenfeld. Artifact support for Ada. Technical report, Intermetrics, Inc., April 1991.

[Car92] W.E. Carlson. Prototyping: An engineering approach to software development. In *Proceedings of the DARPA Software Technology Conference*, April 1992.

[COH92] K. Chen, M. Odersky, and P. Hudak. Parametric type classes. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 170–181. ACM, June 1992.

[GH90] J. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of Symposium on Logic in Computer Science*, pages 333–343. IEEE, June 1990.

[GH91] J. Guzmán and P. Hudak. First-order liveness analysis via type inference. In *XVII Latin American Informatics Conference*, July 1991.

[Guz92] J. Guzmán. *On Expressing the Mutation of State in a Functional Programming Language*. PhD thesis. Yale University, Department of Computer Science, expected 1992.

[HF92] P. Hudak and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

[HPJWe92] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[HR92] P. Hudak and D. Rabin. Mutable abstract datatypes—or—how to have your state and munge it too. Technical Report YALEU/DCS/RR-914, Yale University, Department of Computer Science, July 1992.

[Kar90] M. Karr. Transaction graphs: A sketch formalism for activity coordination. Technical report, Software Options, Inc., March 1990.

[Kar92a] M. Karr. A modest proposal for the unification of data transport across time, distance, language, hardware, and schema evolution: `function` is the thesean answer to procrustean types. Technical report, Software Options, Inc., October 1992.

[Kar92b] M. Karr. A programmable graphics editor based on Emacs, the X toolkit, and Ghostscript. Technical report, Software Options, Inc., November 1992.

[Kar92c] M. Karr. Transaction graph editor. Technical report, Software Options, Inc., December 1992.

[KC90] M. Karr and T. Cheatham. A solution to the ISPW-6 software process modeling example. Technical report, Software Options, Inc., November 1990.

[KH91] M. Karr and P. Hudak. A kernel family and the common prototyping system. Technical report, Software Options, Inc. and Yale University, Department of Computer Science, January 1991.

[mhd92a] MHDL feasibility study. Technical Report IR-MD-157, Intermetrics, Inc., August 1992.

[mhd92b] MHDL preliminary design rationale. Technical Report IR-VA-031, Intermetrics, Inc., November 1992.

[mhd92c] MHDL language reference manual. Technical Report IR-VA-025, Intermetrics, Inc., November 1992.

[Mor92] R. Morris. Activity description cookbook. Technical report, Software Options, Inc., April 1992.

[ORH92] M. Odersky, D. Rabin, and P. Hudak. Call-by-name, assignment, and the lambda-calculus. Technical Report YALEU/DCS/RR-929, Yale University, Department of Computer Science, October 1992. To appear in Proc. 20th ACM Symposium on Principles of Programming Languages.

[Pet92] J. Peterson. Yale Haskell user's guide. Technical Report YALEU/DCS/RR-935, Yale University, Department of Computer Science, August 1992.